Architecture overview:

Angular is a framework for building client applications in HTML and either JavaScript or a language like TypeScript that compiles to JavaScript.

**Modules:**

Angular apps are modular and Angular has its own modularity system called Angular modules or NgModules. Every Angular app has at least one Angular module class, the root module, conventionally named AppModule.

Decorators are functions that modify JavaScript classes.

[NgModule](https://angular.io/api/core/NgModule) is a decorator function that takes a single metadata object whose properties describe the module. The most important properties are:

* declarations - the view classes that belong to this module. Angular has three kinds of view classes: [components](https://angular.io/guide/architecture#components), [directives](https://angular.io/guide/architecture#directives), and [pipes](https://angular.io/guide/pipes).
* exports - the subset of declarations that should be visible and usable in the component [templates](https://angular.io/guide/architecture#templates) of other modules.
* imports - other modules whose exported classes are needed by component templates declared in this module.
* providers - creators of [services](https://angular.io/guide/architecture#services) that this module contributes to the global collection of services; they become accessible in all parts of the app.
* bootstrap - the main application view, called the root component, that hosts all other app views. Only the root module should set this bootstrap property.

Angular modules vs. JavaScript modules

The Angular module — a class decorated with @NgModule — is a fundamental feature of Angular.

JavaScript also has its own module system for managing collections of JavaScript objects. It's completely different and unrelated to the Angular module system.

In JavaScript each file is a module and all objects defined in the file belong to that module. The module declares some objects to be public by marking them with the export key word. Other JavaScript modules use import statements to access public objects from other modules.

Components:

A component controls a patch of screen called a view.

Template:

A template is a form of HTML that tells Angular how to render the component.

Metadata:

Metadata tells Angular how to process a class.

providers: array of **dependency injection providers** for services that the component requires. This is one way to tell Angular that the component's constructor requires a HeroService so it can get the list of heroes to display.

Data binding:

In two-way binding, a data property value flows to the input box from the component as with property binding. The user's changes also flow back to the component, resetting the property to the latest value, as with event binding.

Directive

**A directive is a class with a @Directive decorator**. A component is a directive-with-a-template; a @Component decorator is actually a @Directive decorator extended with template-oriented features.

Two other kinds of directives exist: structural and attribute directives.

**Structural** directives alter layout by adding, removing, and replacing elements in DOM. - \*ngFor, \*ngIf

An **Attribute** directive changes the appearance or behavior of a DOM element. In templates they look like regular HTML attributes, hence the name. – ngModel

Service:

Service is a broad category encompassing any value, function, or feature that an application needs.

A service is typically a class with a narrow, well-defined purpose. It should do something specific and do it well.

Dependency injection

Dependency injection is a way to supply a new instance of a class with the fully-formed dependencies it requires. Most dependencies are services. Angular uses dependency injection to provide new components with the services they need.

You've learned the basics about the eight main building blocks of an Angular application:

* [Modules](https://angular.io/guide/architecture#modules)
* [Components](https://angular.io/guide/architecture#components)
* [Templates](https://angular.io/guide/architecture#templates)
* [Metadata](https://angular.io/guide/architecture#metadata)
* [Data binding](https://angular.io/guide/architecture#data-binding)
* [Directives](https://angular.io/guide/architecture#directives)
* [Services](https://angular.io/guide/architecture#services)
* [Dependency injection](https://angular.io/guide/architecture#dependency-injection)

Template and Data binding

Lifecycle Hooks

Angular offers **lifecycle hooks** that provide visibility into these key life moments and the ability to act when they occur.

Component Interaction

Directives overview

There are three kinds of directives in Angular:

1. Components—directives with a template.
2. Structural directives—change the DOM layout by adding and removing DOM elements.
3. Attribute directives—change the appearance or behavior of an element, component, or another directive.

*Components* are the most common of the three directives.

*Structural Directives* change the structure of the view. Two examples are [NgFor](https://angular.io/guide/template-syntax" \l "ngFor) and [NgIf](https://angular.io/guide/template-syntax" \l "ngIf)

*Attribute directives* are used as attributes of elements. The built-in [NgStyle](https://angular.io/guide/template-syntax" \l "ngStyle) directive in the [Template Syntax](https://angular.io/guide/template-syntax) guide, for example, can change several element styles at the same time.

Pipe:

A pipe takes data as input and transforms it to a desired output. [DatePipe](https://angular.io/api/common/DatePipe), [UpperCasePipe](https://angular.io/api/common/UpperCasePipe), [LowerCasePipe](https://angular.io/api/common/LowerCasePipe), [CurrencyPipe](https://angular.io/api/common/CurrencyPipe), and [PercentPipe](https://angular.io/api/common/PercentPipe)

Form

<https://stackoverflow.com/questions/39142616/angular2-forms-api-template-driven-or-reactive>

**Template Driven Forms Features**

* Easy to use
* Suitable for simple scenarios and fails for complex scenarios
* Similar to angular 1
* Two way data binding(using [(NgModel)] syntax)
* Minimal component code
* Automatic track of the form and its data(handled by Angular)
* Unit testing is another challenge

**Reactive Forms Features**

* More flexible, but needs a lot of practice.
* Handles any complex scenarios.
* No data binding is done(Immutable data model preferred by most developers)
* More component code and less HTML markup
* Reactive transformations can be made possible such as
  + Handling a event based on a debounce time
  + Hanlding events when the components are distinct until changed
  + Adding elements dynamically
* Easier Unit testing